View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Ghent University Academic Bibliography

Systematic functional design of XML tools

Hannes Verlinde*
hannes.verlinde@UGent.be

Raymond Boute
boute@intec.UGent.be

INTEC, Ghent University, Belgium

Abstract

We present a systematic approach for developing software, using Haskell as a programming language and Funmath as a formal specification and verification language. We argue that a separate formal specification of the desired functionality helps bridging the gap between the initial informal specification and the implementation. We illustrate the application of our approach for the development of a structural, DTD dependent XML editor.

1 Introduction

In software development, the use of a formal modeling language can bridge the gap between an informal specification and a final implementation, thus enhancing productivity, reliability and understanding. In recent years, there has been a substantial amount of research on object oriented analysis and design using UML. The general consensus now seems to be that any software engineer using an object oriented approach can benefit from adopting (semi-)formal methods like UML.

Because of the near-declarative nature and algebraic properties of functional programs, using formal methods with functional programming might sound like overkill to some. It is our opinion however, that using the right specification language in combination with prototyping in a functional programming language like Haskell can likewise produce an additional enhancement in productivity and reliability. UML is not a serious candidate, as it is too strongly biased towards the object oriented paradigm and thereby restrictive. Instead we advocate the use of Funmath, a truly declarative specification and verification language, with an extensive collection of calculation rules in a broad application area. We have applied this approach [22] in a context where Haskell had already proven to be a most suitable language, i.e., the development of XML tools.

In section 2 we give a brief overview of XML, Funmath and Haskell. In section 3 we outline our approach and the main advantages. Section 4 illustrates the application of this approach for the systematic development of a graphical, structural

*Research funded by a Ph.D grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

XML editor. In section 5 we formulate some conclusions and give an overview of related and future research.

2 Background

2.1 XML

A markup language allows to add structure to the content of any document. The *Extensible Markup Language* (XML) [5] is a subset of the more complicated *Standard Generalized Markup Language* (SGML) that can be implemented relatively easily while retaining enough flexibility to cover a wide range of applications. XML allows to annotate text files with markup tags that explicitly describe the structural content. While the syntax is simple and fixed, the author is free to use his own tags, making XML both a uniform and a flexible language.

The XML syntax is based on a simple hierarchical view of the data. A *well-formed* XML document, i.e., a document that conforms to all the XML syntax rules, can be represented by a tree structure. Logically it consists of an optional prologue and epilogue and a single rooted tree containing the basic data elements of the document as nodes. The data of each element consists of other elements or character data, preceded by a start tag and followed by an end tag: `<element>data</element>`. An empty element is an element without any data, which can be denoted by `<element></element>` or simply `<element />`. It is possible to add attributes to any element: `<element attribute="value">data</element>`.

```
<?xml version="1.0"?>
<movie genre="Action">
  <title>The Untouchables</title>
  <director>Brian De Palma</director>
  <year>1987</year>
  <actor>Kevin Costner</actor>
  <actor>Sean Connery</actor>
  <actor>Robert De Niro</actor>
</movie>
```

Figure 1: A well-formed XML document

Fig. 1 gives an example of a well-formed XML document. Note that the markup tags are chosen to be self-explanatory. The corresponding element tree is pictured in Fig. 2. A more extensive introduction to XML can be found in [15, 18] and detailed reference material in [5, 10, 24].

A *Document Type Definition* (DTD) [5] is used to define the structure of a certain class of XML documents by specifying a fixed set of markup tags, the corresponding attributes and the required order of the elements. An example of a valid DTD is given in Fig. 3. For every kind of basic data element, there is a corresponding `!ELEMENT`-tag in the DTD, specifying its structure. A *terminal*

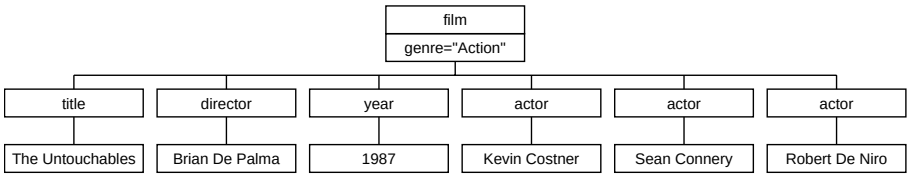

Figure 2: The element tree

```
<!ELEMENT movie (title, director, year?, actor*)>
<!ATTLIST movie genre (Action | Drama | Comedy | Thriller)#REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT director (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT actor (#PCDATA)>
```

Figure 3: A DTD for movie documents

element is an element which cannot contain any other elements, i.e., it can only contain character data or it must be empty. In Fig. 3, `title`, `director`, `year` and `actor` are defined as terminal elements containing only character data. For a *non-terminal element*, the occurrence and the order of the elements corresponding with the child nodes in the element tree can be specified. For instance, the first line in Fig. 3 specifies that every `movie` element must contain a sequence consisting of a `title` element, a `director` element, an optional (as indicated by "`?`") `year` element and zero or more (as indicated by "`*`") `actor` elements. Attributes are either required, implied or fixed, as specified in the corresponding `!ATTLIST`-tag. The value of an attribute usually consists of character data, but it is also possible to define an enumeration type, as for attribute `genre` in Fig. 3.

While it is perfectly possible to write XML documents without any DTD, the resulting documents can only be checked for well-formedness, i.e., syntactical correctness. The use of a DTD allows to check whether an XML document conforms to additional, application-dependent specifications. A well-formed XML document is said to be *valid* with respect to a given DTD if it conforms to all the specifications of that DTD. While the XML syntax rules are fixed, every application can define its own DTD, meaning validity or grammatical correctness is a more dynamic and flexible concept than well-formedness.

XML Schema (XSD) [8] is a more recent alternative for DTD's which is rapidly gaining popularity in the XML community. XSD is based on the same idea of grammatical correctness, but an XML schema has greater expressive power than a DTD. Also, an XML schema is a well-formed XML document, while DTD's use a separate syntax. For technical reasons (including reuse of existing tools), our software is DTD oriented and makes no use of XSD. The underlying ideas, however, are based on the general concept of grammatical correctness.

2.2 Funmath

Functional Mathematics [3] is an approach to structure formalisms by conceiving mathematical objects as functions whenever convenient — which is quite more often than common practice reflects. Here we only provide a rather compact summary of the key conventions used in the sequel. Full details are given in [3].

A binding $v : X \mathbin{\wedge} p$ ($\mathbin{\wedge} p$ optional) introduces the identifier v, specifying $v \in X \wedge p$. For instance, $n : \mathbb{N} \mathbin{\wedge} n/2 \in \mathbb{N}$ declares n to be an even natural number.

A function f is fully defined by its domain $\mathcal{D}\, f$ and its mapping (image for every domain element). Some functions can be denoted by an abstraction of the form *binding . expression*. Writing f for $v : X \mathbin{\wedge} p \,.\, e$, the domain axiom is $d \in \mathcal{D}\, f \equiv d \in X \wedge p[^v_d$ and the mapping axiom is $d \in \mathcal{D}\, f \Rightarrow f\, d = e[^v_d$. Here $e[^v_d$ is e with d substituted for v. For instance, $n : \mathbb{N} \,.\, 2 \cdot n$ doubles naturals. Another example is the constant function definer $^\bullet$ with $X^\bullet e = v : X \,.\, e$ (taking v not free in e).

Predicates are $\mathbb{B}$-valued functions ($\mathbb{B} = \{0, 1\}$). The quantifiers $\forall$ and $\exists$ are defined as predicates over predicates: $\forall P \equiv P = \mathcal{D}\, P^\bullet 1$ and $\exists P \equiv P \neq \mathcal{D}\, P^\bullet 0$. Writing predicates as abstractions conveniently yields familiar expressions such as $\forall P \equiv \forall x : \mathcal{D}\, P \,.\, P\, x$ and $\forall x : \mathbb{R} \,.\, x^2 \geq 0$. An extensive set of rules making formal calculation with quantifiers convenient and useful in everyday practice is given in [3].

Generic functionals are general-purpose operators on arbitrary functions. Here we only introduce a few from a rather extensive collection [4]. The *range* operator $\mathcal{R}$ has axiom $e \in \mathcal{R}\, f \equiv \exists x : \mathcal{D}\, f \,.\, f\, x = e$. Using $\{\text{—}\}$ as a synonym for $\mathcal{R}$ synthesizes set notations such as $\{m : \mathbb{N} \mid m < n\}$, with the general convention that $x : X \mid p$ is shorthand for the abstraction $x : X \mathbin{\wedge} p \,.\, x$. Expressions like $\{e, e', e''\}$ also have their usual meaning. The function arrow operator $\rightarrow$ is defined by $f \in X \rightarrow Y \equiv \mathcal{D}\, f = X \wedge \mathcal{R}\, f \subseteq Y$, for any sets X and Y. The generic set filtering operator $\downarrow$ is defined by $X \downarrow P = \{x : X \cap \mathcal{D}\, P \mid P\, x\}$ for any set X and predicate P. We write X_P as a shorthand for $X \downarrow P$. The function composition operator $\circ$ is defined by $f \circ g = x : \mathcal{D}\, g \mathbin{\wedge} g\, x \in \mathcal{D}\, f \,.\, f\, (g\, x)$. Note that we do not restrict the argument functions but, instead, precisely define the domain of the result.

A sequence or tuple is any function with domain $\square\, n$, with $n : \mathbb{N}$ or $n := \infty$. The block operator $\square$ is defined by $\square\, n = \{m : \mathbb{N} \mid m < n\}$, so $\square\, 0 = \emptyset$, $\square\, 2 = \mathbb{B}$ and $\square\, \infty = \mathbb{N}$. The length operator $\#$ is defined by $\#\, x = n \equiv \mathcal{D}\, x = \square\, n$. The empty tuple is ε. The shift operator σ has axioms $\#\, (\sigma\, x) = \#\, x - 1$ and $\sigma\, x\, n = x\, (n + 1)$. The prefixing operator $\succ\!-$ is defined by $\#\, (a \succ\!- x) = \#\, x + 1$ and $(a \succ\!- x)\, i = (i = 0\, ?\, a \mathbin{\mathrm{\dagger}} x\, (i - 1))$ (the conditional of the form $c\, ?\, b \mathbin{\mathrm{\dagger}} a$ being defined by $(a, b)\, c$ or, alternatively, $0\, ?\, b \mathbin{\mathrm{\dagger}} a = a$ and $1\, ?\, b \mathbin{\mathrm{\dagger}} a = b$). The postfixing operator $-\!\prec$ is defined similarly. An array of length n over set A is a function of type $\square\, n \rightarrow A$, written A^n. The set of lists over A is $\bigcup n : \mathbb{N} \,.\, A^n$, written A^*, and A^+ is the set of nonempty lists.

Finally, the generalized functional Cartesian product $\bigtimes$ [4] is defined for any family T of sets by $\bigtimes T = \{f : \mathcal{D}\, T \rightarrow \bigcup T \mid \forall x : \mathcal{D}\, T \,.\, f\, x \in T\, x\}$. Observe that, for sets A and B, $\bigtimes (A, B) = A \times B$ and $\bigtimes (A \bullet B) = A \rightarrow B$. If T is an abstraction of the form $x : A \,.\, B$, where B may depend on x, then $\bigtimes x : A \,.\, B$ is often written as $A \ni x \rightarrow B$.

2.3 Haskell

A distinguishing characteristic of functional programming languages, as opposed to traditional imperative programming languages, is the stronger focus on *what* is to be computed, not *how* it should be computed [16], and constitutes a first step towards declarativity. Because of the referential transparency and rigorous control of side effects, it is far easier to prove certain properties for functional programs than for imperative ones. Functional programming languages have many other serious advantages over imperative languages. We mention some of the specific advantages of the purely functional language Haskell for writing (large) software systems [16]:

- Substantially increased programmer productivity.
- Shorter, clearer, and more maintainable code.
- Fewer errors, higher reliability.
- A smaller *semantic gap* between the programmer and the language.
- Shorter lead times.

For certain application areas, like multimedia, there are additional advantages [13]. Here we will focus on the development of XML tools. An XML document can be represented by a tree, and functional programming languages are strong in working with recursive data types and hierarchically structured data in general. The simple typing language of DTD's can be embedded in the richer Haskell type system by interpreting the declaration of an element in a DTD as a data type declaration in Haskell [23]. A sequence of child nodes can be represented by a Haskell product type and a list of alternatives by a sum type. An optional node is represented by a `Maybe` type and repetition of elements by a Haskell list. There is an obvious mapping between the XML documents that are valid with respect to a given DTD and the values of the corresponding Haskell data type. This kind of shallow embedding leads to very elegant programs for processing XML, e.g., checking XML documents for well-formedness and validity is already implicitly implemented by the Haskell type checking mechanism. A thorough introduction to Haskell can be found in [2, 13, 14].

Because of the many strengths of Haskell, which has even been dubbed an *executable specification language*, using a separate formal specification language might seem superfluous or at least redundant at first. There is, however, still a considerable difference with Funmath, which is a purely declarative formalism that offers an abstract reasoning level completely independent of any implementation. Funmath also offers a wide range of formal proof techniques and calculation rules, which allows proving properties in a universal mathematical framework.

3 Systematic approach

We suggest the following general purpose methodology for the development of software, using Haskell as a programming language and Funmath as a formal specification and verification language:

- Providing an informal but (preferably) detailed specification of the desired new functionality.
- Providing a formal specification of existing software that will be used in the final implementation. This should be a relatively easy task because the expressive power of a formalism like Funmath subsumes any existing programming languages.
- Providing a formal specification of the desired new functionality based on both the informal description of the new functionality and the formal description of the existing software. Because of the expressive power and universal applicability of Funmath, a semantic gap between the informal and the formal specification can be avoided to a large extent.
- Verifying any required properties of the functionality based on the formal specification. Within Funmath, there is an extensive collection of formal calculation rules available for this task.
- Implement a prototype based on the formal specification. The gap between a formal specification in Funmath and an initial implementation in Haskell is relatively small, because both languages are based on the mathematical concept of a function and many executable Funmath constructs have an equivalent representation in Haskell.
- Optimize the prototype implementation if necessary.

It is our opinion that applying the above approach can further strengthen the aforementioned advantages of using Haskell. For instance, the semantic gap between programmer and programming language will be even smaller due to the intermediate formal specification step. As a consequence, the source code will be clearer and more maintainable. Whenever a problem occurs, the developer can always fall back on the formal definitions. Programmer productivity will increase because mistakes in the design can be caught at an early stage of the development, before any actual implementation takes place. Finally, the reliability will be higher because more properties can be more easily verified using the formal calculation rules in Funmath. Additional advantages of our approach are that the formal specification in Funmath serves as a platform independent reference for the implemented functionality and forces the developer to give a precise description of the desired functionality in an early stage of the development. We will now illustrate our approach by providing a case study, i.e., the systematic development of a graphical, structural XML editor.

4 Development of a structural XML editor

4.1 Informal specification

An XML document is a text document with a clean separation between the actual content and the markup tags. It is possible to edit XML documents with an ordinary text editor, but this approach brings along considerable repetitive work, is

very sensitive to typing errors and requires technical knowledge of the XML syntax rules. To ensure the syntactical and grammatical correctness of an XML document, it is possible to perform an automatic check after every modification to the document. With a specialized XML editor, XML documents can be edited in a more intuitive, user-friendly manner. Modifications to the document are restricted to ensure that the well-formedness and validity of the document can never be invalidated, alleviating the need for a separate check routine. We also want the final editor to be *structural*, meaning that the building blocks are the basic syntactic XML constructs, not the individual characters of the text file.

Whether an XML document is well-formed or not depends on a fixed set of XML syntax rules, but the concept of validity only makes sense with respect to a given specification. Hence the functionality of the editor depends at least partially on a given DTD. Informally, the intended basic functionality of such a DTD dependent XML editor consists of the following functions:

- Opening and saving a document. Opening an existing XML document only succeeds if the document is well-formed and valid with respect to the given DTD. When opening a new XML document, the result is not an empty document, but rather a *minimal* document that conforms to the DTD.
- Displaying the content of a document. This function will generate a text representation of the content of the XML document, along with additional information that can be used for navigation and lay-out.
- Editing a document. This is the core function of the editor, which consists of three separate phases:
  - The user navigates in the document and selects a location.
  - The possible modifications for that location are presented to the user.
  - The user selects a modification which is then executed.

  For a given location in the document, the number of possible modifications is usually rather low, so it makes sense to present them to the user in a selection menu (Fig. 4). To edit character data in the document, we will provide a simple text subeditor.

This informal description of the desired functionality is too vague to be implemented directly. The required level of detail will usually become clear when constructing the formal specification and resolving certain ambiguities along the way.

4.2 Formal specification in Funmath

Formally, an editor for a given data type can be characterized as a piece of software that allows to manipulate values of that data type. In our case, the data type is the class of XML documents that are valid with respect to a given DTD, hence the resulting editor is DTD dependent. To avoid redundant work, the final implementation will make use of the Haskell toolkit HaXml [23], which consists of a

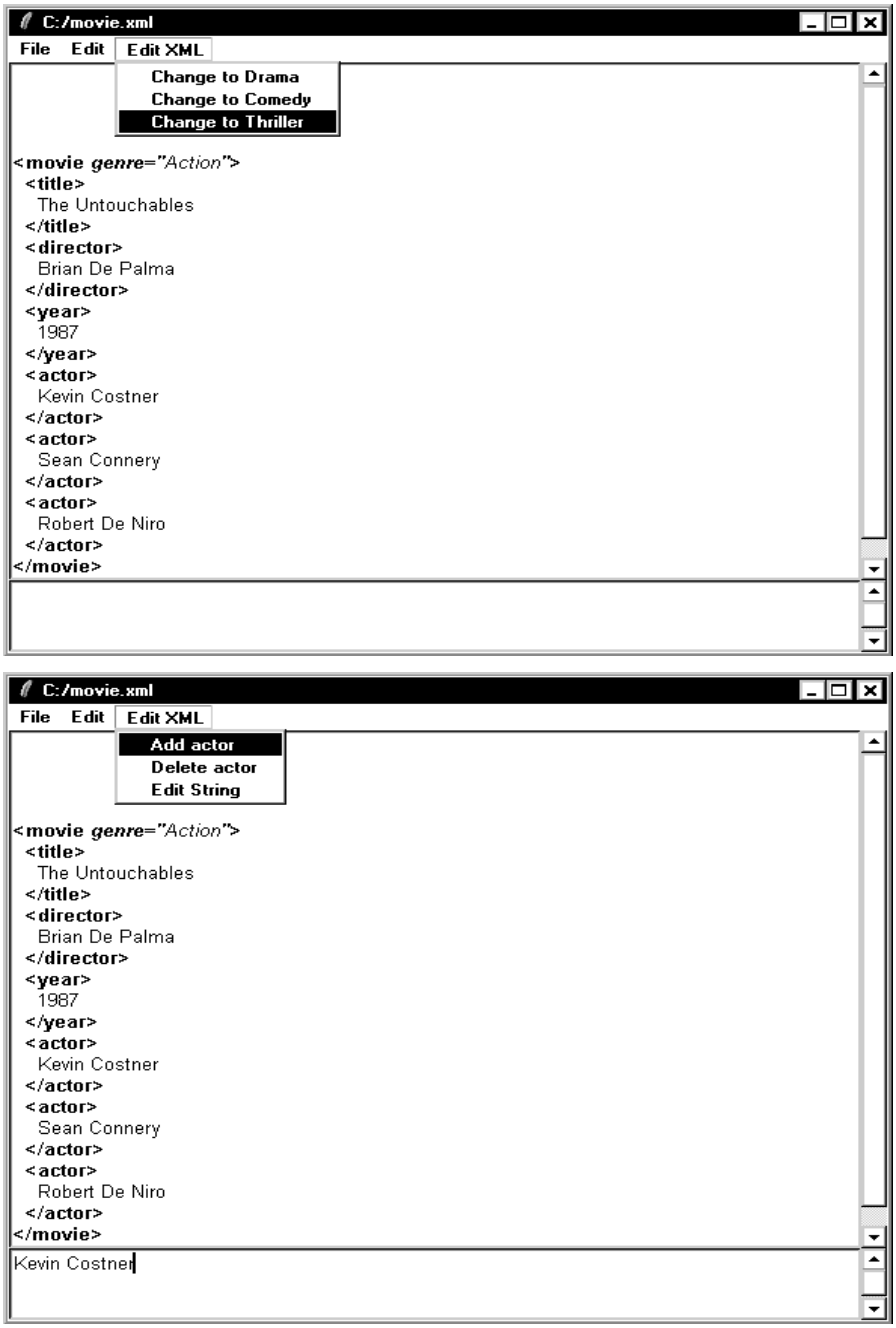

Figure 4: The editor in action

number of tools to aid in processing XML. For our purposes, the tool DtdToHaskell is especially useful. The input for this application is a valid DTD and the output is Haskell source code, consisting of a Haskell data type declaration and functions to read in and write out values of that data type. These values correspond to the XML documents that are valid with respect to the DTD. In order to implement the required new functionality, we will simply extend DtdToHaskell to generate additional functions, e.g., a function that returns a default value of the data type, which corresponds to a minimal XML document that conforms to the DTD. The resulting application, i.e., an extended version of DtdToHaskell, is not yet an editor, but generates Haskell source code that contains all the required functionality. Additionally we will implement a graphical user interface that links all these functions together.

The Funmath specification of our application consists of a set of formal function definitions. Here we restrict ourselves to discussing the typing of these functions. The full function definitions along with a more detailed discussion can be found in [22].

The input for our application is a DTD, i.e., a text file of type $\mathbb{A}^*$. Obviously not all possible text files conform to the DTD syntax rules, and parsing will only succeed if the input is a valid DTD. This means that, at least conceptually, there is an implicit validity function available to decide whether a text file is a valid DTD. Every valid DTD will be parsed to a list of *type definitions*. Each type definition corresponds to a single `!ELEMENT`-tag in the original DTD. We may further assume that the first type definition in the list corresponds to the root element.

$$\begin{aligned} \mathsf{validdtd} &: \mathbb{A}^* \to \mathbb{B} \\ \mathsf{parsedtd} &: (\mathbb{A}^*)_{\mathsf{validdtd}} \to \mathsf{Typedef}^+ \end{aligned}$$

Recall that X_P is shorthand for $X \downarrow P$. All DTD dependent functionality can now be modeled by functions taking a list of type definitions as a parameter. The remaining functionality of the original DtdToHaskell tool is specified by the following DTD dependent functions:

$$\begin{aligned} \mathsf{toType} &: \mathsf{Typedef}^+ \to \mathcal{T} \\ \mathsf{valid} &: \mathsf{Typedef}^+ \to \mathbb{A}^* \to \mathbb{B} \\ \mathsf{open} &: \mathsf{Typedef}^+ \ni t \to (\mathbb{A}^*)_{\mathsf{valid}\ t} \to \mathsf{toType}\ t \\ \mathsf{save} &: \mathsf{Typedef}^+ \ni t \to \mathsf{toType}\ t \to (\mathbb{A}^*)_{\mathsf{valid}\ t} \end{aligned}$$

The function toType associates a rather simple, recursive data type with every list of type definitions. In the final implementation, this data type will be a Haskell data type, but here we do not have this restriction so we simply use the type universe $\mathcal{T}$ as codomain. Values of the resulting data type correspond to XML documents that are valid with respect to the list of type definitions and hence with respect to the original DTD.

The function open is a parser for this class of XML documents. Again there is an implicit validity function (conveniently called valid), because parsing only succeeds for XML documents that are both syntactical and grammatical correct. The type

of the function open can be interpreted as follows: for any list of type definitions t, any text document that is valid with respect to t will be mapped to a value of the data type associated with t.

The function save is the semi-inverse of open, meaning that the composition (open $\circ$ save) yields the identity function but the converse is not necessarily true. The XML syntax allows a certain freedom, e.g., regarding the use of whitespace, meaning that two (slightly) different text files might represent the same XML document. Such differences will however be eliminated when both text files are first opened and then saved again.

To specify the core functionality of the editor, we need a few additional DTD dependent functions:

$$
\begin{array}{ll}
\mathsf{new} & : \mathsf{Typedef}^+ \ni t \rightarrow \mathsf{toType}\ t \\
\mathsf{showXml} & : \mathsf{Typedef}^+ \ni t \rightarrow \mathsf{toType}\ t \rightarrow (\mathbb{A}^* \times \mathbb{N}^* \times \mathsf{ContentType})^+ \\
\mathsf{alternatives} & : \mathsf{Typedef}^+ \ni t \rightarrow \mathsf{toType}\ t \rightarrow \mathbb{N}^* \rightarrow (\mathbb{A}^+)^* \\
\mathsf{update} & : \mathsf{Typedef}^+ \ni t \rightarrow \mathsf{toType}\ t \rightarrow \mathbb{N}^* \rightarrow \mathbb{A}^+ \rightarrow \mathsf{toType}\ t
\end{array}
$$

The function new returns a default value for the data type associated with a list of type definitions. This value corresponds to a minimal XML document where all optional elements are left out, all lists of elements are empty, all enumerations are set to their first value and all required character data is set to "-".

To display an XML document on screen, the function showXml will be used. This function returns a list of tuples representing the contents of the document. Every tuple consists of a text string, a navigation code and a content type. Each text string represents a small part of the actual content. The content type assigned to each text string will be used to implement a simple syntax highlighting mechanism and is either *element*, *attribute*, *enum* or *string*. The navigation code associated with each text string is a sequence of naturals of type $\mathbb{N}^*$ and will be used to navigate through the document. Whenever a user selects a certain text string on screen, either by clicking or by moving the cursor, the associated navigation code will tell us the exact location in the document. To construct the navigation code, the n child nodes of every node in the element tree are marked with the numbers 0 to $n-1$, hence a unique sequence of naturals corresponds with each path from the root to a node.

Given a document and a navigation code, the function alternatives returns a list of possible modifications for that specific location in the document. The possibilities are represented by non-empty text strings of type $\mathbb{A}^+$, e.g., "Add actor" or "Change to Thriller", which can then be composed to form a selection menu to be presented to the user. Once the user has chosen a specific modification, the function update will return the modified value of the (same) data type.

Besides the DTD dependent functions that make up the core of the editor, there are also a few more general editor functions that are independent of any specific DTD.

Example: Consider the *undo* function, which allows the user to restore a document to a previous state when an unwanted modification has been made. Assume

we have an undo-buffer u of finite size n to store recently modified document values and a redo-buffer r of equal size, allowing the user to navigate in both directions between a limited number of recent document values. We obviously need to alter the update function to take these buffers into account. Note that both buffers u and r have type $\bigcup i : \square\,(n+1)\,.\,A^i$, assuming the type of document values is A. An update function that modifies the value *old* into the value *new* can be modeled as follows:

$$\begin{aligned}&\textbf{def}\ \mathit{update} : A \times A^* \times A^* \to A \times A^+ \times A^*\\ &\textbf{with}\ \mathit{update}\,(\mathit{old}, u, r) = (\mathit{new}, \mathsf{take}\ n\,(\mathit{old} \mathbin{>\!\!-} u), \varepsilon)\end{aligned}$$

The function take is simply an auxiliary function used to enhance readability:

$$\begin{aligned}&\mathsf{take} : \mathbb{N} \to A^* \to A^*\\ &\mathsf{take}\ 0\,x = \varepsilon\\ &\mathsf{take}\ n\,\varepsilon = \varepsilon\\ &\mathsf{take}\ n\,(a \mathbin{>\!\!-} x) = a \mathbin{>\!\!-} \mathsf{take}\,(n-1)\,x\end{aligned}$$

After any update, the new undo buffer is constructed by prefixing the old document value to the existing buffer, and retaining the first n values of the result. The new redo buffer will be empty because a redo action only makes sense if preceded by one or more undo actions without any updates in between. The actual undo and redo functions can be modeled in a similar fashion:

$$\begin{aligned}&\textbf{def}\ \mathit{undo} : A \times A^+ \times A^* \to A \times A^* \times A^+\\ &\textbf{with}\ \mathit{undo}\,(\mathit{old}, u, r) = (u\,0, \sigma\,u, \mathit{old} \mathbin{>\!\!-} r)\\ &\textbf{def}\ \mathit{redo} : A \times A^* \times A^+ \to A \times A^+ \times A^*\\ &\textbf{with}\ \mathit{redo}\,(\mathit{old}, u, r) = (r\,0, \mathit{old} \mathbin{>\!\!-} u, \sigma\,r)\end{aligned}$$

The above definitions may appear rather trivial, but formally proving a few expected properties reveals some of the subtleties (e.g. why is the use of take required in the definition of *update* but not in the definitions of *undo* and *redo*?). Some of the properties we expect to hold are:

$$\begin{aligned}(\mathit{undo} \circ \mathit{update})\,(\mathit{old}, u, r)\,0 &= \mathit{old}\\ (\mathit{redo} \circ \mathit{undo})\,(\mathit{old}, u, r) &= (\mathit{old}, u, r)\\ (\mathit{undo} \circ \mathit{redo})\,(\mathit{old}, u, r) &= (\mathit{old}, u, r)\end{aligned}$$

With the extensive collection of Funmath calculation rules for tuples and sequences at our disposal, proving these properties is a simple exercise (elaborated here just

by way of illustration):

$$
\begin{array}{lrl}
 & & (\mathit{redo} \circ \mathit{undo})\,(\mathit{old}, u, r) \\
= & \langle \text{definition } \circ \rangle & \mathit{redo}\,(\mathit{undo}\,(\mathit{old}, u, r)) \\
= & \langle \text{definition } \mathit{undo} \rangle & \mathit{redo}\,(u\,0, \sigma\,u, \mathit{old} \succ\!\!- r) \\
= & \langle \text{definition } \mathit{redo} \rangle & ((\mathit{old} \succ\!\!- r)\,0, u\,0 \succ\!\!- \sigma\,u, \sigma\,(\mathit{old} \succ\!\!- r)) \\
= & \langle (a \succ\!\!- x)\,0 = a \rangle & (\mathit{old}, u\,0 \succ\!\!- \sigma\,u, \sigma\,(\mathit{old} \succ\!\!- r)) \\
= & \langle \sigma\,(a \succ\!\!- x) = x \rangle & (\mathit{old}, u\,0 \succ\!\!- \sigma\,u, r) \\
= & \langle x = (x\,0 \succ\!\!- \sigma\,x) \rangle & (\mathit{old}, u, r)
\end{array}
$$

The formalization outlined above can serve as a platform independent specification for the functionality of an (XML) editor, potentially valuable to anyone developing such an application. Proving properties using Funmath is relatively easy due to the available formal proof techniques and allows design errors to be caught at an early stage of development, which avoids later additional implementation costs.

4.3 Implementation in Haskell

Implementing our editor prototype in Haskell based on the Funmath specification is rather straightforward. For each formal function definition we implement a corresponding Haskell function.

There are a few subtle but important semantic issues to be dealt with in this process, but we do not necessarily consider this to be a disadvantage. Making the semantic differences between the formal specification and the implementation explicit provides additional insight that helps bridging the gap.

As stated before, the main application generates Haskell source code that will be called from the graphical user interface linking everything together. The formal Funmath functions are all DTD dependent, i.e., they take a list of type definitions as a primary argument. The corresponding Haskell functions are also DTD dependent, but it is unnecessary, and even undesirable from a performance point of view, to have a list of type definitions as an explicit argument. Surely, even though the functionality for e.g. creating a new document definitely depends on the DTD, it would be a bad idea to evaluate the type definitions every time a new document is created. Instead we can evaluate these type definitions just once, i.e., when the source code is generated, and generate a specific set of functions based on this evaluation. The resulting functions are DTD dependent, but this dependency is built in, i.e., a function application does not require an evaluation of the DTD. The generated Haskell functions are simply partial applications of the corresponding Funmath functions and this shift in semantics does not imply any additional problems.

The internal structure of the extended DtdToHaskell application is pictured in Fig. 5. The dashed lines represent the newly implemented functionality. For a given DTD, the generated source code consists of a Haskell data type and a set of functions declaring this data type to be an instance of the classes `XmlContent` and `Editable`.

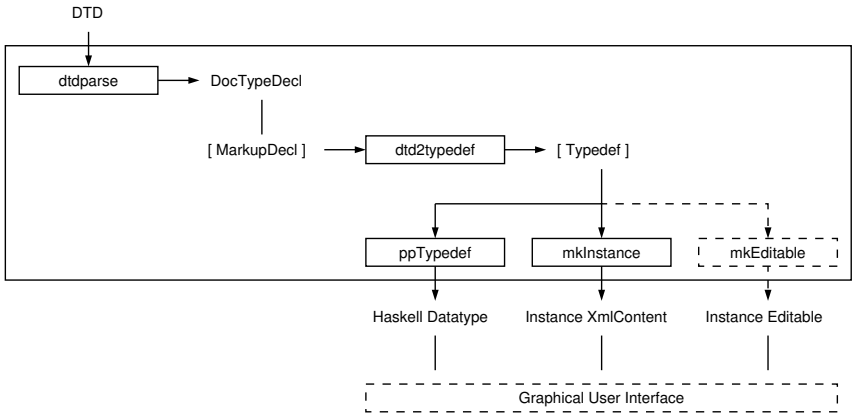

Figure 5: Inside view of the software

The function `ppTypedef` is an implementation of the formally defined toType function that generates a Haskell data type based on a list of type definitions. Applying this function to the type definitions associated with the DTD from Fig. 3 yields the following result, which should be fairly self-explanatory to readers familiar with Haskell:

```
data Movie = Movie Movie_Attrs Title Director (Maybe Year) [Actor]
data Movie_Attrs = Movie_Attrs { movieGenre :: Movie_Genre }
data Movie_Genre = Movie_Genre_Action  | Movie_Genre_Drama  |
                   Movie_Genre_Comedy  | Movie_Genre_Thriller
newtype Title = Title String
newtype Director = Director String
newtype Year = Year String
newtype Actor = Actor String
```

The function `mkInstance` takes a list of type definitions as its argument and generates the corresponding functions used to implement `readXml` and `writeXml`, i.e., the Haskell counterparts of the formally defined open and save functions:

```
readXml  :: (XmlContent a) => FilePath -> IO a
writeXml :: (XmlContent a) => FilePath -> a -> IO ()
```

We extend DtdToHaskell with the function `mkEditable` in order to generate the DTD dependent functions used to implement the core functionality of the editor:

```
new          :: (Editable a) => a
showXml      :: (Editable a) => a -> [ContentElement]
alternatives :: (Editable a) => a -> [Int] -> [String]
update       :: (Editable a) => a -> [Int] -> String -> a
```

Note the resemblance in typing with the (partial applications of) the formal Funmath functions with the same name. A similar resemblance can be found in the actual function definitions, omitted here for brevity's sake, further justifying the separate formal specification phase.

4.4 The graphical user interface

The extended DtdToHaskell application basically generates a Haskell data type based on a given DTD, along with functions to edit values of this data type. It is the task of the (graphical) user interface (GUI) to link the functionality together and make it accessible to the user. There are several GUI toolkits available for Haskell. We opted for FranTk [19], a toolkit built upon Tcl/Tk, which is a largely platform independent scripting language for developing GUI's. FranTk allows to implement a GUI in Haskell while employing a declarative style of programming using the GUI monad, an extension of the standard IO monad [17]. Values of type `GUI a` represent actions that return a value of type `a` and that may have a side effect on the user interface. FranTk is based on the same mathematical foundations as the *Functional Reactive Animation* language (FRAN) [7]. Concepts like *listeners* and *events*, which obey certain algebraic laws, can be used to construct a model of a time dependent system which can be formally verified. FranTk also provides a set of primitive widgets which can be combined into more powerful constructs. More details and an overview of our editor's GUI source code can be found in [22].

5 Conclusions, related and future work

It is our opinion that an explicit formal specification helps bridging the gap between an initial informal specification and a final implementation. Obviously, the implementation itself is also a formal specification, but programming languages are often too restrictive to express certain concepts in the most intuitive way.

A truly declarative formalism like Funmath does not have the same restrictions, while implementing a prototype in Haskell based on a specification in Funmath is usually rather straightforward. Combining Funmath and Haskell allowed us to develop a working prototype of a DTD dependent XML editor with a modest amount of resources.

By using the toolkits HaXml and FranTk, we could restrict the implementation of the new functionality to 350 lines of source code, with an additional 200 lines for the graphical user interface. Developing an actual production type would further involve an additional fine-tuning phase in order to obtain a more efficient and user-friendly software product.

The HaXml toolkit implements two complementary approaches to combine Haskell and XML [23]. The first approach consists in defining a generic combinator library used to process XML documents, without considering the grammatical correctness of the documents. The second approach generates a Haskell data type based on a given DTD, effectively embedding the typing language of DTD's in the

Haskell type system. This way the Haskell type checker can be used to check XML documents for grammatical correctness.

A similar, more recently developed toolkit is the Haskell XML Toolbox [20], which also includes an XPath [6] interpreter. Danny Van Velzen has developed an implementation in Haskell of an XML library, an XPath interpreter and an XSLT [1] processor [21]. Generic Haskell [11] is an extension of Haskell which allows defining generic functions, i.e., functions taking arbitrary data types as arguments. Generic Haskell can be used to develop DTD dependent XML tools in an elegant way [12]. Recently the core functionality of an XML editor was implemented in Generic Haskell [9].

Even though the XML community currently advocates the adoption of XML Schema, many in the Haskell community still prefer DTD's because they can be easily mapped to Haskell data types. It would be interesting to find out to what extent combining Haskell and XML Schema actually requires a deeper and hence less elegant embedding.

Another promising research direction consists in embedding a Haskell implementation of XSLT in our DTD dependent XML editor. In a more general context, it would be interesting to explore the similarities between generic combinator libraries written in Haskell — as used for parsing, pretty printing, web scripting, XML processing, etc. — and the generic functionals that are defined in Funmath.

References

[1] S. Adler, A. Berglund, J. Caruso, S. Deach, T. Graham, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, J. Richman and S. Zilles. "Extensible Stylesheet Language (XSL) Version 1.0". W3C Recommendation, 2001.
(`http://www.w3.org/TR/xsl/`)

[2] R. Bird. *Introduction to Functional Programming using Haskell: second edition.* Prentice Hall, 1998.

[3] R. Boute. *Functional Mathematics: a Unifying Declarative and Calculational Approach to Systems, Circuits and Programs — Part I.* Course notes, Ghent University, 2002.

[4] R. Boute. "Concrete Generic Functionals: Principles, Design and Applications", in: Jeremy Gibbons and Johan Jeuring, eds. *Generic Programming.* Kluwer, 2003. pp. 89-119.

[5] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler and F. Yergeau. "Extensible Markup Language (XML) 1.0 (Third Edition)". W3C Recommendation, 2004.
(`http://www.w3.org/TR/REC-xml/`)

[6] J. Clark and S. DeRose. "XML Path Language (XPath) Version 1.0". W3C Recommendation, 1999.(`http://www.w3.org/TR/xpath/`)

[7] C. Elliott and P. Hudak. "Functional Reactive Animation". *Proceedings ACM SIGPLAN International Conference on Functional Programming*, 1997. pp. 263-273.

[8] D. C. Fallside. "XML Schema Part 0: Primer". W3C Recommendation, 2001. (`http://www.w3.org/TR/xmlschema-0/`)

[9] P. Hagg. "A framework for developing generic XML tools". Master's thesis, Utrecht University, 2002.

[10] E. R. Harold. *XML Bible*. Hungry Minds, 1999.

[11] R. Hinze and J. Jeuring. "Generic Haskell: Practice and Theory". *Lecture Notes of the Summer School on Generic Programming*, Springer-Verlag, 2003.

[12] R. Hinze and J. Jeuring. "Generic Haskell: Applications". *Lecture Notes of the Summer School on Generic Programming*, Springer-Verlag, 2003.

[13] P. Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, New York, 2000.

[14] P. Hudak, J. Peterson and J. Fasel. "A Gentle Introduction to Haskell". 2000. (`http://www.haskell.org/tutorial/`)

[15] D. Hunter, J. Rafter, J. Pinnock, C. Dix, K. Cagle and R. Kovack. *Beginning XML*. Wrox Press, 2001.

[16] "A short introduction to Haskell". Based on a paper by Simon Peyton Jones. 2001. (`http://www.haskell.org/aboutHaskell.html`)

[17] S. P. Jones. "Tackling the Awkward Squad: monadic input/output, concurrency, exceptions and foreign-language calls in Haskell". *Engineering theories of software construction*, IOS Press, 2001. pp. 47-96.

[18] S. S. Laurent. *XML: A Primer*. Hungry Minds, 1999.

[19] M. Sage. "FranTk — A Declarative GUI System for Haskell". *Proceedings ACM SIGPLAN International Conference on Functional Programming*, 2000.

[20] M. Schmidt. "Design and Implementation of a validating XML parser in Haskell". Master's thesis, Wedel University, 2002.

[21] D. van Velzen. "An XSLT implementation in Haskell". Master's thesis, Utrecht University, 2001.

[22] H. Verlinde. "Systematisch ontwerp van XML-hulpmiddelen in een functionele taal". Master's thesis, Ghent University, 2003.

[23] M. Wallace and C. Runciman. "Haskell and XML: Generic Combinators or Type-Based Translation?". *Proceedings International Conference on Functional Programming*, 1999.

[24] H. Williamson. *XML: The Complete Reference*. McGraw-Hill Osborne Media, 2001.